Ejecución y adaptación de trazas de juegos para la automatización de pruebas

Trabajo de Fin de Grado en el
Doble Grado en Ingeniería Informática y Matemáticas

Jennifer Hernández Bécares

Dirigido por el Doctor
Pedro Pablo Gómez Martín

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Junio 2015

Ejecución y adaptación de trazas de juegos para la automatización de pruebas

Trabajo de Fin de Grado realizado por

Jennifer Hernández Bécares

Dirigido por el Doctor

Pedro Pablo Gómez Martín

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Junio 2015

A mis padres

Agradecimientos

A todos los que siguen queriendo ser diferentes y luchan contra aquellos que desean que seamos iguales.

Albert Espinosa

Muchas personas han pasado por mi vida desde que empecé el Doble Grado. No ha sido un camino fácil, pero esta memoria prueba que gracias a ellos he conseguido llegar al final.

Me gustaría agradecer en primer lugar a mis padres, por su incondicional apoyo de principio a fin, a pesar de las innumerables veces que he querido darme por vencida.

A Pedro Pablo, por acceder a dirigir este proyecto y soportar reuniones improvisadas todas las semanas. A Marco, por demostrar que las clases no tienen por qué ser siempre aburridas, por gastar sus tardes entrenándonos para la Swerc durante tres años, por los cafés, sus consejos de futuro y por confiar en mí desde el principio. Gracias a los dos por hacer que nuestro paso por la universidad haya sido memorable.

A mis compañeros y profesores, por todo lo que me han enseñado a lo largo de estos cinco años.

Por último, a Luisma, compañero de proyecto, de prácticas y de fatigas. Gracias por cada momento, por ayudarme a mejorar día a día y animarme siempre a seguir adelante.

Sin todos vosotros, nada de esto habría sido posible. Gracias.

Resumen

A computer would deserve to be called intelligent if it could deceive a human into believing that it was human.

Alan Turing

Tradicionalmente, las empresas de videojuegos y los desarrolladores invierten mucho tiempo y recursos en probar que los videojuegos siguen teniendo el comportamiento esperado después de realizar ciertos cambios en él. Este trabajo lo llevan a cabo lo que conocemos como testeadores humanos, que juegan una y otra vez los mismos juegos y niveles con el objetivo de detectar fallos. Debido a la naturaleza de los videojuegos, la automatización de las pruebas mediante soluciones como tests de unidad es prácticamente imposible.

En este trabajo se propone una forma alternativa de realizar tests, que se basa en la reproducción de trazas adaptadas que se han generado y guardado anteriormente al jugar el testeador humano. Con este método, si los cambios en el juego son mínimos, si son cambios a nivel de programación, etcétera, se podrá comprobar que su comportamiento es correcto sin ningún esfuerzo adicional.

Para llevar a cabo nuestro objetivo, proponemos un sistema externo de generación, salvado y reproducción de trazas, así como también un método de adaptación de las mismas que utiliza el razonamiento basado en casos y la inteligencia artificial ya existente en el videojuego usado como base (Blázquez Checa et al., 2013-2014) para llevar a cabo las acciones necesarias para completar los objetivos sin ningún error.

Palabras clave: arquitecturas de juegos, automatización de pruebas, grabación de trazas, pruebas unitarias, videojuegos

Abstract

A computer would deserve to be called intelligent if it could deceive a human into believing that it was human.

Alan Turing

Traditionally, game industry companies and developers spend a lot of time and resources in testing videogames to prove they keep having the same expected behaviour after performing some changes at it. This job is carried out by what we know by human testers, who play the same games and levels time and time again with the purpose of detecting errors. Because of the nature of videogames, automation of testing with solutions like unit tests is almost impossible.

Here we propose an alternate form of making and passing tests, based on the reproduction of adjusted traces that where generated and saved before, when the human tester was playing the game. With this method, if the changes in the game are minimum or if they are programming-only changes then checking that the behaviour is correct can be done without any additional effort.

In order to complete our goal, we propose an external system that generates, saves and reproduces traces, as well as a method for adapting these traces. This method uses case-based reasoning and the artificial intelligence that previously existed in the game used (Blázquez Checa et al., 2013-2014) to accomplish the actions needed to complete all the objectives without making any mistakes.

Keywords: game architecture, test automation, trace recording, unit testing, videogames

Índice

Índice de figuras

Capítulo 1

Introducción

Isn't it nice to think that tomorrow is a new day with no mistakes in it yet?

L. M. Montgomery

1.1. Antecedentes y motivación del proyecto

Con el crecimiento progresivo del software se ha puesto de manifiesto la importancia de incorporar el *testing* en el ciclo de vida de desarrollo. Llevar a cabo pruebas que demuestren la corrección de un sistema implementado se convierte en algo crucial, ya que la aparición de *bugs* se incrementa con el aumento de la cantidad de código utilizado. Por ello, es necesario asegurarse de que cualquier software cuenta con pruebas exhaustivas diseñadas específicamente para cada tipo de proyecto.

La aparición de diferentes tipos de videojuegos (la mayoría de ellos con un nivel de complejidad bastante elevado) descubre la necesidad de enfocar el testing de otro modo. Los test que se han llevado a cabo tradicionalmente están basados en demostrar la corrección de la implementación de módulos o funciones específicos del juego. Sin embargo, esto no es suficiente. Los videojuegos son mucho más que eso. Llevando a cabo pruebas unitarias comunes, es imposible probar la jugabilidad del resultado o asegurar que la experiencia de usuario es óptima.

Debido a que los videojuegos son elementos interactivos, la entrada de los mismos resulta muy compleja. Esto provoca que además las diferentes "salidas" generada por un videojuego no sean fáciles de comparar, ya que su ejecución no genera un fichero con valores que se puedan comprobar de forma sencilla. Es por esto que para poder comprobar la corrección de un videojuego se tienen que llevar a cabo pruebas manuales que verifiquen cada uno de los aspectos del mismo.

Por otro lado, la posibilidad de modificar el mapa de los niveles de un juego desde ficheros externos introduce la posibilidad de que un diseñador pueda realizar cambios sin contar con los programadores. Al llevar a cabo dichas modificaciones, será necesario disponer de un sistema que compruebe que los niveles no se han roto. Como hemos dicho antes, los test de unidad no son suficiente para realizar todas las comprobaciones necesarias, por lo que en este trabajo planteamos una solución que permitirá automatizar la realización de test de unidad de videojuegos incluso cuando los mapas se han alterado por un diseñador.

1.2. Objetivos y plan de trabajo

Debido a la necesidad de crear una nueva forma de realizar test aplicada a videojuegos, se propone en este proyecto llevar a cabo una grabación y reproducción de trazas. Grabar las trazas de los niveles del juego consistirá en almacenar información sobre las acciones realizadas por el jugador cuando prueba el juego en primera instancia para posteriormente poder utilizarlas. Usando esa información, se podrán reproducir las trazas con el objetivo de realizar distintas comprobaciones. Entre ellas se encuentran demostrar que un nivel del juego no se rompe después de que el diseñador del mapa introduzca cambios en él o que el videojuego sufra modificaciones no visibles en la implementación, como mejoras en la eficiencia o en la búsqueda de caminos de la Inteligencia Artificial del juego.

En vista de lo anterior, se propone el siguiente plan de trabajo:

- Diseño de un sistema de testing avanzado que pueda aplicarse a más aspectos de los videojuegos.
- Elección de un videojuego que permita introducir funcionalidad de forma sencilla.
- Estudio de la arquitectura del videojuego elegido.
- Introducción en el juego de un nuevo componente que permita grabar trazas del juego.
- Generación de la información y elección del formato de las trazas generadas. Almacenamiento de la información en un fichero.
- Adaptación de las trazas para que puedan ser reproducidas posteriormente.
- Introducción de un *parser* que lea de forma sencilla la información contenida en el fichero generado.
- Estudio de las redes de Petri para que puedan ser aplicadas a la reproducción de trazas.

- Reproducción de trazas utilizando redes de Petri temporizadas con el objetivo de llevar a cabo pruebas que comprueben la corrección del videojuego al realizar cambios en él.
- Análisis de los requisitos necesarios en un videojuego para que sea factible grabar y reproducir trazas para realizar test.
- Estudio de las posibles mejoras y ampliaciones de este proyecto en el futuro.

1.3. Contenido de la memoria

Capítulo 2: *Testing.* Este capítulo introduce los conceptos de testing y test de unidad. Se habla del origen de los test de unidad, así como de sus características y principales ventajas. Se termina el capítulo hablando de las limitaciones que las pruebas unitarias tienen cuando los sistemas aumentan su complejidad y cuando el sistema a probar es un videojuego.

Capítulo 3: *Arquitectura de videojuegos.* En este capítulo se define el concepto de motor de un videojuego, hablando sobre los diferentes tipos de videojuegos y los respectivos requisitos tecnológicos que tiene cada uno de esos motores. Posteriormente se habla de arquitecturas basadas en jerarquías de clases y los problemas que pueden surgir al usarlas. Por último, se introducen las arquitecturas basadas en componentes en contraposición a las basadas en jerarquías y se define una forma de comunicación específica para componentes que se basa en el envío de mensajes o eventos.

Capítulo 4: *Grabación, adaptación y reproducción de trazas.* En este capítulo se explica de forma general el sistema que hemos implementado para grabar y reproducir trazas de juegos. Se diferencia entre la grabación por entrada directa del usuario (teclado y ratón) y la grabación de acciones a alto nivel, que más tarde se utilizará para adaptar y reproducir trazas del juego. Se introducen las redes de Petri temporizadas como modelo de apoyo a la reproducción de acciones en el videojuego. Se termina el capítulo hablando de los requisitos necesarios en un juego para que la reproducción de trazas que hemos implementado funcione correctamente y se pueda utilizar sin apenas cambios.

Capítulo 5: *Time and Space.* Este capítulo define un dominio del videojuego Time and Space, que se ha utilizado para probar los componentes introducidos en el capítulo 4. Finalmente se cuenta cómo se ha aplicado la grabación y reproducción de trazas en este videojuego en concreto, mostrando un ejemplo de las trazas generadas para su reproducción

posterior e introduciendo al lector en la automatización de pruebas de videojuegos.

Capítulo 6: *Conclusiones y trabajo futuro.* En este capítulo se resumen los puntos más importantes comentados anteriormente en los capítulos previos. Por último, una vez probada la utilidad de la nueva forma de realizar test se habla de posibles ampliaciones del trabajo llevado a cabo en este proyecto.

Apéndice A: *Implementación de los módulos extra.* En este apéndice se puede consultar parte del código generado para llevar a cabo la grabación y reproducción de trazas de juegos en Time and Space.

Capítulo 2

Testing

I'm supposed to be a scientific person,
but I use intuition more than logic in
making basic decisions.

Seymour R. Cray

Resumen: En este capítulo se explican los conceptos de testing y pruebas unitarias, hablando también de las herramientas existentes para llevar a cabo la tarea de validación y verificación del software. Se concluye el capítulo realizando una pequeña discusión sobre las limitaciones de las pruebas unitarias cuando éstas se tienen que aplicar a videojuegos.

2.1. ¿Qué es el testing? ¿En qué consisten los test de unidad?

El estándar (IEEE Std 829-1998, 1998) define el testing como el proceso de analizar un elemento del software para detectar las diferencias entre las condiciones existentes y las requeridas (es decir, los fallos) y evaluar las características de dicho elemento. Algunas de estas características son el rendimiento, la funcionalidad, la usabilidad, la precisión, la arquitectura de red, la fiabilidad del sistema o la portabilidad.

Los test de unidad o pruebas unitarias son procedimientos de desarrollo en los cuales los programadores diseñan y programan las pruebas al mismo tiempo que desarrollan el software. Dichos test son pruebas cortas y sencillas que comprueban el funcionamiento de módulos particulares del código, tales como una clase o una función.

Para cada módulo o característica del software se definen reglas por las cuales se determinará si el test de unidad se pasa o si por el contrario falla.

Para determinar el resultado de la prueba se definen los resultados esperados, que posteriormente se contrastarán con los obtenidos con el objetivo de saber qué errores hay y de dónde provienen.

Es importante que en los documentos de pruebas se especifique qué cosas se quieren probar y cómo se probarán, pero también se deben identificar qué módulos importantes no se probarán y por qué no se hará.

2.2. Origen del concepto de testing

El primero en referirse al testing como tal y diferenciarlo de la depuración de programas fue Glenford Myers, en Myers (1979). En este libro, el autor define depuración (debugging) como el proceso que se debe seguir una vez se haya ejecutado con éxito un caso de prueba. Para él los casos de prueba que tienen éxito son aquellos que consiguen encontrar un fallo.

En contraposición con la definición propuesta en 2.1 por el estándar IEEE, Myers define el testing como el proceso de ejecutar un programa con el objetivo de encontrar errores en él. Habla de la importancia de entender el testing de esta forma, ya que piensa que las personas están muy orientadas a conseguir objetivos. Esto puede llevar a intentar demostrar que los programas no tienen errores, en lugar de intentar demostrar que sí los tienen. Se tenderá a elegir datos de entrada que no hagan fallar nuestro programa y los resultados de la fase de testing no serán válidos, ya que los errores surgirán en una fase posterior y deberán corregirse mediante depuración.

2.3. ¿Por qué surgen los test de unidad?

El objetivo de los test de unidad es demostrar que las partes individuales de un programa son correctas. Para pasar ciertas pruebas se requiere que cada parte del código cumpla unos estrictos requisitos que se han establecido previamente. El cumplimiento de esas restricciones impuestas de forma necesaria para poder pasar los test proporciona beneficios. Los principales se listan a continuación:

- **Detección temprana de problemas:** Llevar a cabo test de unidad permite encontrar problemas en las primeras fases del ciclo de desarrollo del software. Encontrar fallos de programación en etapas tempranas disminuye los costes que surgen al tener que arreglar los errores posteriormente.

- **Facilidad de cambio:** La existencia de test permite a los programadores actualizar el código o mejorar las librerías usadas y seguir teniendo una forma de comprobar que dichas modificaciones no afectan al correcto funcionamiento del módulo a probar.

- **Simplicidad de integración:** Gracias a la existencia de test se puede demostrar que las partes individuales de un programa funcionan correctamente y una vez que se tiene esa certeza se pueden llevar a cabo pruebas que progresivamente integren los módulos anteriores de forma más sencilla.

Por todo lo anterior, en el proceso de desarrollo se vuelve esencial introducir pruebas mediante las cuales se tenga la certeza de que la aplicación que se está desarrollando produce los resultados esperados y que son correctos en todo momento, con la posibilidad de limitar, detectar y eliminar errores fácilmente y con costes mucho menores.

2.4. Características de las pruebas unitarias

Las pruebas unitarias deberían tener ciertos atributos, a saber: deben ser automatizables, completas, repetibles e independientes. Además, las pruebas unitarias deberán considerarse igual de importantes que el propio código, documentándolas y llevándolas a cabo con la mayor exactitud posibles.

Una vez conseguido lo anterior, algunos de los objetivos buscados al automatizar pruebas son los siguientes (véase el Capítulo 3 de Meszaros (2007)):

- Deben ayudar a mejorar la calidad.
- Deben reducir los riesgos, y no introducir aumentarlos.
- Deben ser fáciles de lanzar.
- Deben ser fáciles de escribir y mantener.
- Deben requerir un mínimo mantenimiento mientras el sistema evoluciona a su alrededor.

Todos los atributos anteriormente mencionados contribuyen a que el desarrollo de pruebas se realice de un modo más sencillo, que permita mejorar los sistemas a probar. Deben ser fáciles de repetir, consiguiendo siempre resultados equivalentes para los mismos datos de entrada.

2.4.1. Herramientas de pruebas

Algunas de las herramientas y bibliotecas usadas para llevar a cabo pruebas unitarias son las siguientes:

- **JUnit**: Conjunto de bibliotecas creado por Erich Gamma y Kent Beck. Está hecho específicamente para realizar pruebas de aplicaciones escritas en Java. Ha sido de gran importancia para los procesos de desarrollo guiados por pruebas. Además, existen complementos para Eclipse

o Netbeans que facilitan la creación de las clases que coordinarán las pruebas, permitiendo que el desarrollador se centre en la prueba y los resultados obtenidos.

- **CppUnit**: Herramienta para pruebas unitarias en C++ distribuida bajo licencia LGPL en varias distribuciones Linux.

- **SUnit**: Conjunto de herramientas para escribir y lanzar casos de prueba en Smalltalk, un lenguaje de programación orientado a objetos. SUnit fue escrito originalmente en 1994 por Kent Beck.

- **xUnit**: Nombre colectivo que se le da al conjunto de entornos para la realización de test de unidad que se derivan de SUnit.

2.5. Limitaciones de las pruebas unitarias

A medida que los sistemas aumentan su tamaño y su complejidad, las pruebas unitarias se vuelven más difíciles de diseñar y desarrollar (Mellon, 2006). Probar todas las funciones y características del software programado se vuelve una tarea muy difícil. Existen varios ejemplos que demuestran las limitaciones de las pruebas unitarias. Por ejemplo, cuando los módulos tienen un gran tamaño, la preparación necesaria para poder integrar y lanzar los test de unidad requiere mucho esfuerzo. Otro de los ejemplos más claros de esta limitación es el del desarrollo de videojuegos en línea y multijugador. Al aumentar el número de jugadores de forma sustancial, las pruebas no son capaces de detectar todos los problemas que pueden surgir y es necesario seguir realizando desarrollo significativo para corregirlo durante amplios periodos de tiempo, que en algunos casos han llegado a superar los diez años.

Los videojuegos de estas características se vuelven muy difíciles de especificar y depurar, ya que puede existir entrada múltiple de datos (correspondiente a cada uno de los jugadores), no determinismo y una gran cantidad de procesos corriendo al mismo tiempo. Para realizar pruebas en este nivel de complejidad, es necesario diseñar toda la arquitectura del sistema de una forma concreta en las primeras etapas del proceso de desarrollo para que soporte la integración de pruebas. Realizar estas pruebas de forma automática puede ayudar a tratar con los problemas que surgen, y también hace que sea más fácil encontrar errores y volver a reproducirlos. Además, para llevar a cabo pruebas automatizadas no será necesario que haya alguien supervisando la ejecución de las mismas. Los datos de entrada y salida de las pruebas se decidirán inicialmente, pero después de lanzarlas no será necesaria la presencia de nadie. Gracias a la automatización se podrán ejecutar múltiples pruebas a la vez, incluso en distintas máquinas y con datos de entrada muy distintos unos de otros.

A pesar de todo lo anterior, hay errores que no se podrán encontrar simplemente con la realización de test automáticos, ya que muchos de los fallos que se encuentran cuando los usuarios prueban el sistema se debe a que sus acciones pueden resultar imprevisibles. Además, surgen varios problemas relacionados con la estructura de pruebas de un sistema. En primer lugar, el diseño del software se ve modificado con el único objetivo de realizar test posteriormente y la implementación se ve deteriorada al intentar integrar pruebas unitarias. Por otro lado, los test no se usan siempre de la forma correcta y en el momento más adecuado: hay que tener en cuenta que cada tipo de prueba tiene un momento óptimo para realizarse. En las primeras etapas y en las etapas intermedias las pruebas deberán ser muy generales. Se comprobará que el camino crítico sigue funcionando y que con una sola persona o con un número muy limitado de ellas se puede jugar. En etapas finales del desarrollo se buscará probar cada detalle de la funcionalidad y realizar pruebas de carga de los servidores. La escalabilidad es una característica muy difícil de probar. Por último, no suele haber ingenieros expertos destinados a diseñar y realizar pruebas, sino que se prefiere que ingenieros no expertos programen test básicos y posteriormente destinar dinero y recursos en contratar a diferentes personas que ayudarán a probar el sistema y depurar los errores encontrados.

A pesar de que es una práctica cara tener en cuenta todas las dificultades que puedan surgir y actuar para prevenir los errores, los costes alternativos son mucho más altos y no siempre funcionan. Es por esto que los test de unidad suelen ser una buena solución para demostrar que la mayor parte de módulos de un videojuego son correctos. Sin embargo, un videojuego es mucho más que un conjunto de módulos correctos. Por ejemplo, la experiencia del usuario es imposible de probar mediante test de unidad. En el caso de juegos, probarlos completamente significa disponer de varias consolas, conexiones de red, ordenadores o televisiones. Por todo ello, es necesario realizar muchos test antes de realizar las pruebas con usuarios reales, ya que además de perder dinero se les hará perder mucho tiempo si los resultados no son satisfactorios. Esto nos lleva a buscar una nueva forma de realizar pruebas que sea más adecuada en el caso de los videojuegos.

En el capítulo 3 se explicarán las dos arquitecturas de videojuegos más usadas tradicionalmente y la comunicación mediante paso de mensajes que se usa en las arquitecturas basadas en componentes. Después, en el capítulo 4 se explicará la forma en la que hemos implementado para grabar y reproducir trazas de videojuegos, que más tarde se podrá usar para llevar a cabo test específicos para juegos. Estos nuevos test proporcionarán mejores resultados en videojuegos que los test de unidad, y permiten automatizar la difícil tarea de depurar juegos tal y como se conoce actualmente.

Capítulo 3

Arquitectura de videojuegos

Nothing that's worthwhile is ever easy.
Remember that.

Nicholas Sparks

Resumen: Este capítulo introduce el concepto de motor de un videojuego y los requisitos tecnológicos que se necesitan en función de los tipos de juegos que se estén implementando. A continuación se presenta la arquitectura de videojuegos basada en componentes como alternativa a la arquitectura basada en jerarquías de clases. Por último, se cuenta la comunicación mediante paso de mensajes aplicada a la arquitectura basada en componentes.

3.1. Introducción

Las arquitecturas de juegos han sufrido numerosos cambios en los últimos años. Por un lado, los motores del juego se han creado tradicionalmente de forma específica para cada género, ya que cada uno de ellos necesita unos requerimientos tecnológicos que pueden ser muy distintos de los que necesiten el resto de géneros. Por otro, las arquitecturas han ido evolucionando de forma que sea más fácil incorporar funcionalidad nueva, es decir, que el motor del juego resulte reutilizable y mantenible.

En este capítulo se introducen los conceptos de motor del juego y los dos tipos de arquitectura de juegos más conocidos, exponiendo las ventajas e inconvenientes de cada uno de ellos y contando finalmente cual de ellas resulta más útil para los propósitos de este trabajo.

3.2. Motor del videojuego

El término "Motor del videojuego" (Gregory, 2009) surgió a mediados de los años 90 en referencia a los juegos de disparo en primera persona, como Doom, de la compañía Id Software. Este videojuego poseía una arquitectura con una separación muy bien definida entre los componentes centrales del juego (sistema de detección de choques, sistema de renderizado gráfico o sistema de audio) y los mundos o las reglas del juego. Esta separación demostró ser muy valiosa, por lo que los desarrolladores comenzaron a crear motores del juego comunes que prácticamente no sufrían cambios al usarse en un nuevo juego. De esta forma, los juegos serían distintos entre sí porque cada vez se crearían nuevos mundos, armas, vehículos o reglas, pero apenas se vería modificado el software central del mismo. La reutilización de motores del juego ha sido una pieza clave en la construcción de los mismos, ya que a pesar de la gran inversión inicial que requiere, sigue siendo mucho más económico que desarrollar de nuevo el motor central cada vez. Debido a la importancia que han cobrado los motores de juegos, la venta de motores que integren toda la funcionalidad requerida se ha convertido en un mercado en sí mismo, permitiendo que los desarrolladores de videojuegos se preocupen únicamente por el desarrollo del juego, y no tanto por el motor que éste utiliza.

Los motores del juego son típicamente específicos para cada género. Pese a haber un alto grado de superposición, existen unos ciertos requerimientos tecnológicos que son particulares para cada uno de los géneros. A continuación se expone una lista no exhaustiva de los géneros más conocidos con los requerimientos asociados al mismo:

- **Juegos de disparos en primera persona (FPS):** Los juegos pertenecientes a este género son de los más difíciles de construir, ya que tienen como objetivo proporcionar al jugador la ilusión de estar inmerso en un mundo muy detallado y realista. Requieren de tecnologías como un renderizado en 3D eficiente, animaciones de alta fidelidad relacionadas con los personajes, un mecanismo de control de la cámara preciso o una inteligencia artificial y animaciones de alta fidelidad para los no jugadores (enemigos y aliados).

- **Juegos de plataformas:** Este nombre se aplica a aquellos juegos de acción en los cuales saltar de plataforma en plataforma es el mecanismo principal del juego. En ellos, el personaje principal no tiene por qué ser particularmente realista. Necesitará tecnología específica para el movimiento del jugador por plataformas, cuerdas, escaleras, etc.

- **Otros juegos en tercera persona:** Son muy parecidos a los juegos en primera persona y necesitan más o menos la misma tecnología. En

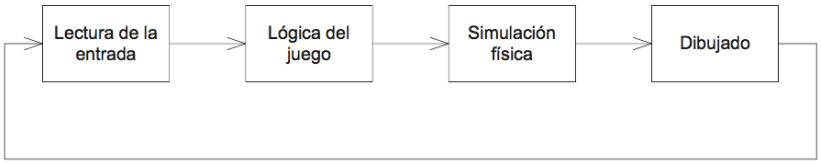


Figura 3.1: Esquema sencillo de bucle principal

cambio, el énfasis se centra en las animaciones y movimientos del personaje principal. Se tendrá que renderizar el cuerpo completo del jugador y no sólo sus brazos y el arma que lleva, por lo que será necesaria una cámara de seguimiento en tercera persona, que controlará el jugador mediante teclado o ratón.

- **Juegos de lucha:** Típicamente son juegos de dos jugadores que se encuentran en una especie de ring de combate. En este género es necesario que las animaciones de lucha sean variadas, la detección de colisiones debe ser precisa y el sistema de entrada del usuario debe detectar combinaciones complejas.
- **Juegos de estrategia en tiempo real (RTS):** En estos juegos, el usuario tiene que construir y desplegar unidades para conseguir ganar a su oponente en batalla. La visión de este mundo suele ser oblicua de arriba a abajo, lo cual permite al desarrollador realizar múltiples optimizaciones en el motor de renderizado.

Los requisitos tecnológicos necesarios para cada tipo dan lugar a la incorporación en el juego de diferentes entidades. Las *entidades* son objetos dinámicos del juego que hacen que el entorno varíe al ver modificadas sus propiedades durante la ejecución del mismo. Pueden estar implementadas en forma de jerarquía o usando componentes. En la sección 3.4 se contará cómo está implementada la arquitectura basada en jerarquías de clases, mientras que en la sección 3.5 se explica de forma resumida cómo funciona una arquitectura basada en componentes. Esta última arquitectura será muy útil debido al solapamiento existente entre los requisitos de los diferentes motores para distintos juegos, ya que permitirá reutilizar partes específicas del código incluso aunque el resultado no se parezca.

3.3. Bucle principal del juego

El bucle principal de un juego dirige la ejecución del mismo, realizando cada tarea para que el resultado visible por el jugador sea óptimo. Es en el bucle principal donde se definen las tareas a realizar y el orden en el cual se llevarán a cabo. En la figura 3.1 se puede ver un esquema con las tareas

que debe realizar el bucle principal, que se detallan a continuación (Gómez-Martín, 2008):

- **Gestión de la entrada del usuario**: la lectura de la entrada suele situarse al principio de la aplicación, ejecutándose una vez por cada vuelta del bucle. Esta lectura puede involucrar acciones tales como comprobar si se ha pulsado alguna tecla o gestionar dispositivos de entrada más complicados. Para garantizar que la aplicación reaccionará de forma consistente en todo el bucle, se suelen analizar todos los dispositivos de entrada a la vez, almacenando el resultado de las lecturas para su consulta posterior por el resto de tareas.

- **Simulación de la lógica**: en esta fase se deciden los comportamientos de los personjes, se ejecutan las acciones de los mismos, se actualiza el estado de los objetos y se lanzan eventos a los que habrá que reaccionar en fases posteriores.

- **Simulación de la física y tratamiento de colisiones**: se encarga de mover los objetos y actualizar los sistemas de partículas y las animaciones de los personajes, así como de gestionar las colisiones provocadas al simular los objetos.

- **Dibujado del mundo**: se utiliza toda la información recogida en las fases previas para actualizar el mundo y dibujar la escena creada a partir de esta.

Generalmente cada vuelta del bucle principal tiene como resultado la creación de un nuevo *fotograma*, que puede tener una duración variable o una duración fija. Si la duración es variable, el tiempo entre fotogramas varía dependiendo de la situación del juego. Por ejemplo, si el juego se encuentra en un momento que requiere gran cantidad de cálculo, disminuyen los fotogramas por segundo. En caso de que la duración del fotograma sea variable, el número por segundo es siempre el mismo.

Por otro lado, para la lógica del juego se pueden usar dos formas distintas de simulación del tiempo: paso de tiempo variable y paso de tiempo fijo. En el caso del tiempo variable, cada vez que una tarea de la lógica se va a ejecutar se comprueba el tiempo que ha pasado desde la última vez que se actualizó. A pesar de ser intuitivo, presenta ciertos problemas: en ciertos juegos no es necesario ejecutar la lógica una vez por fotograma; además, la corrección de errores es difícil, ya que reproducir ejecuciones no siempre produce el mismo resultado. En cambio, si el paso de tiempo es fijo, la simulación se realiza considerando que el tiempo que ha pasado ha sido siempre el mismo.

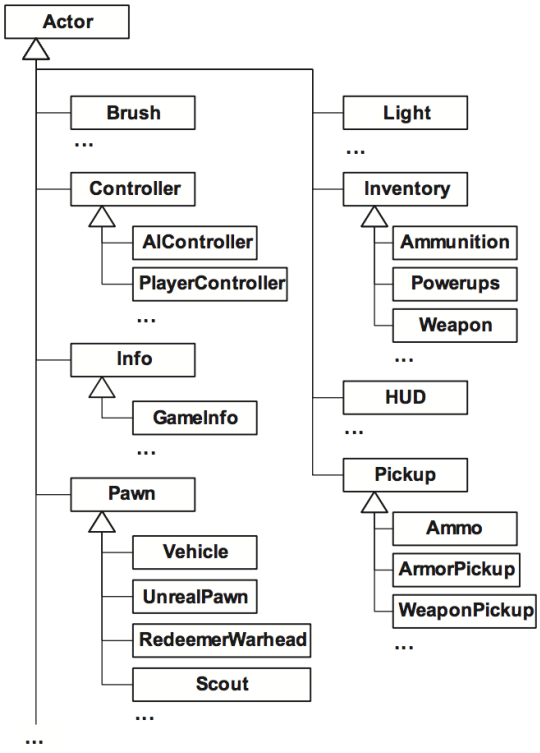


Figura 3.2: Extracto de la jerarquía de clases de Unreal Tournament 2004

3.4. Arquitectura basada en jerarquía de clases

La forma que tradicionalmente se ha usado para representar el conjunto de entidades de un juego ha sido usando jerarquías de entidades (Bilas, 2002). A medida que el desarrollo del juego progresa, es necesario ir añadiendo funcionalidad a las entidades. Esto se puede conseguir encapsulando la nueva funcionalidad dentro de la propia entidad que va a verse modificada o derivando dicha entidad de un objeto que ya tenga. Normalmente, la funcionalidad se introducirá en la jerarquía de clases en algún punto cercano a la raíz. Esto tiene la ventaja de que todas las clases derivadas de la clase raíz podrán utilizar esa funcionalidad, pero también tiene una gran desventaja: las clases derivadas tendrán una gran sobrecarga, puesto que en la mayoría de los casos no necesitarán toda esa funcionalidad adicional. En cambio, si se añaden las nuevas funciones en clases que se encuentran más abajo en la jerarquía, se tenderá a duplicar código, ya que cada objeto que quiera usarlo tendrá que contener la implementación. Como resultado, llegará un momento en el cual sea necesario refactorizar y reestructurar la jerarquía de clases para mover y combinar funcionalidad y evitar los dos problemas mencionados anteriormente.

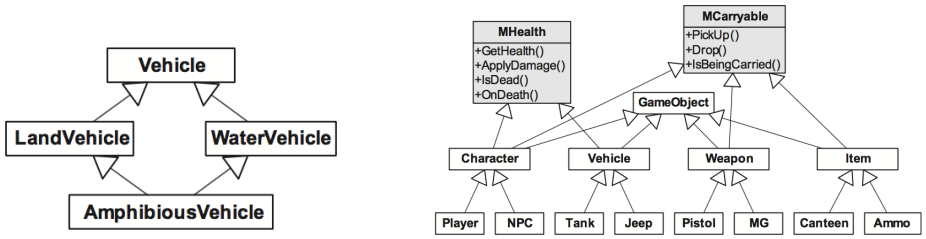
Las jerarquías de clases extensas y demasiado profundas, como la que aparece en la figura 3.2, tienden a causar muchos problemas (Gregory, 2009, p. 716). A medida que una jerarquía se extiende e integra más clases, los problemas se vuelven incluso más extremos. Entre ellos se encuentran los siguientes:

- **Dificultad de comprensión, mantenimiento y modificación de clases**: Para realizar la modificación de cualquier clase, no basta con entender su comportamiento. Es necesario conocer también el comportamiento de las clases padre. Cambiar una función virtual que aparentemente es inofensiva en una clase derivada de otra puede suponer violar las suposiciones hechas en las clases base. Esto puede llevar a la aparición de errores muy difíciles de depurar.
- **Herencia múltiple**: Cuando la herencia múltiple de clases lleva a un objeto a contener varias copias de uno de los miembros de la clase base, se dice que se produce *herencia en diamante*. Este problema se produce cuando dos clases B y C heredan de A, y posteriormente una clase D hereda de B y C. En este caso, si desde la clase D llamamos a un método de A, se producirá ambigüedad en la llamada, ya que estamos heredando ese método de A por dos clases distintas, lo cual es un problema. Un ejemplo de herencia en diamante puede encontrarse en la figura 3.3a, donde se puede ver claramente lo anterior.
- **Clases Mix-In**: Se trata de un ejemplo de herencia múltiple limitada que consiste en heredar de clases "Mix-In", creadas para agrupar funcionalidad y que se introducen en la jerarquía en los lugares que se necesitan. De esta forma, se podrá heredar sólo de una clase padre, pero de tantas clases "Mix-In" como sea necesario. Un ejemplo de jerarquía en la cual se han usado clases Mix-In puede verse en la figura 3.3b.
- **Efecto burbuja**: Inicialmente, las clases raíz de la jerarquía suelen diseñarse para que sean sencillas. Sin embargo, cuando se va añadiendo funcionalidad al juego, surge la necesidad de compartir código entre dos o más clases que no están relacionadas entre sí, causando un efecto burbuja que hace que la funcionalidad se traslade a clases padre.

Por todo ello, es necesario buscar una nueva forma de diseñar la arquitectura que pueda evitar estos problemas, facilitar los diseños y la incorporación de cambios.

3.5. Arquitectura basada en componentes

Una solución a los problemas anteriores es aislar cada una de las características de las clases y convertirlas en clases independientes, haciendo que



(a) Herencia en diamante (b) Jerarquía con clases "Mix-In"

Figura 3.3: Problemas relacionados con las jerarquías de clases

cada una de ellas proporcione un servicio único y bien definido (Gregory, 2009, p. 724). Estas clases independientes se denominan **componentes**, y son piezas de lógica del juego que posteriormente se integrarán para formar entidades. Ejemplos de esos componentes pueden ser la cámara (CCamera), el gráfico (CGraphics), la vida (CLife) o el controlador de la física (CPhysicController).

Tener un diseño basado en componentes permite escoger qué características queremos incluir en cada objeto del juego cuando lo creamos. Además, ayuda a mantener, refactorizar o extender cada una de esas características sin afectar al resto. Las clases individuales son más sencillas de entender y de probar, ya que están desacopladas unas de otras.

Al diseñar una arquitectura basada en componentes, también hay que decidir cómo y cuándo se crearán los componentes y cuándo se van a destruir. Una de las formas de hacer esto es incluir punteros a todos los componentes posibles en la clase base de los objetos. Posteriormente, las clases derivadas podrán crear en su constructor los componentes que necesiten, y éstos serán los componentes pertenecientes a ese objeto.

Una de las dificultades de este diseño surge a la hora de realizar la comunicación entre componentes. Se necesita alguna forma de pasar información de un componente de una clase a componentes de la misma clase o de otra distinta, idealmente de forma eficiente, así como que exista comunicación entre las distintas entidades para que sean notificadas de los cambios importantes en el juego. En la siguiente sección se introduce la comunicación mediante paso de mensajes, desarrollada especialmente para arquitecturas basadas en componentes.

3.6. Comunicación mediante paso de mensajes

Se denomina "mensaje" o "evento" (Gregory, 2009, p. 773) a cualquier cosa de interés que pasa durante el juego. Los juegos necesitan una forma de comunicación entre componentes, es decir, de notificar a los objetos afectados

cuando ocurre un evento y de gestionar las respuestas a esos eventos.

Un mensaje está compuesto de dos partes: el tipo del mensaje y sus argumentos. Los argumentos del mensaje proporcionan información específica de ese mensaje, y dependen del tipo de mensaje que estamos tratando. Informar a una entidad del juego de algo que está ocurriendo es equivalente a enviar un mensaje a dicha entidad.

El tipo y los argumentos de un mensaje pueden encapsularse en un nuevo objeto, y hacerlo proporcionará varios beneficios:

- **Método único para tratar mensajes**: Los mensajes pueden ser representados por la misma clase (o la clase raíz), simplemente creando distintas instancias de ésta. Para ello, sólo se necesitará que los mensajes hereden de una superclase, y de este modo los componentes sólo necesitarán un método para tratar todos los tipos de mensaje. Ya que todos heredarán de una clase, se diferenciarán entre sí gracias al valor de un campo que marcará su tipo.

- **Persistencia**: Los mensajes están representados mediante objetos que contienen datos (el tipo y los argumentos del mensaje), por lo que pueden guardarse en una cola de procesamiento de mensajes que lo tratará más adelante, pueden copiarse o ser reenviados a diferentes receptores.

- **Reenvío a ciegas del mensaje**: Una entidad puede reenviar un mensaje que recibe hacia otra entidad sin saber absolutamente nada sobre el mensaje.

A la hora de tratar los mensajes, lo que se posee es una función ("handler") o varias, que son capaces de manejar los mensajes y responder a ellos. Esta función puede implementarse de dos formas distintas: puede contener una sentencia switch o una cláusula if/else-if que sea capaz de manejar todos los tipos de mensajes que puede recibir o, en cambio, puede haber varias funciones handler que traten cada una de ellas un evento en concreto.

Uno de los problemas que puede aparecer al usar una comunicación por paso de mensajes es el problema de la ineficiencia a la hora de recibir eventos, ya que en principio todas las entidades recibirán el mensaje generado y se lo mandarán a sus componentes. Para evitar esto, puede hacerse que las entidades se registren para recibir ciertos mensajes a los cuales tendrán que responder de algún modo. Así, no será necesario iterar innecesariamente sobre entidades que sabemos de antemano que no están interesadas en el mensaje que estamos enviando.

Con todo lo anterior, disponemos de una arquitectura basada en componentes con un mecanismo de comunicación de paso de mensajes. Esto será útil al introducir nueva funcionalidad en un juego (o un nuevo componente), ya que estará modularizado de tal modo que no habrá que introducir cambios

significativos para que funcione. Sólo habría que crear un nuevo componente y hacer que éste forme parte de la entidad o entidades a las que queremos extender.

Capítulo 4

Grabación, adaptación y reproducción de trazas

Nothing is more creative... nor destructive... than a brilliant mind with a purpose.

Dan Brown

Resumen: En este capítulo se introducen los métodos de grabación y reproducción de trazas que se han implementado aplicados a una arquitectura basada en componentes que utiliza la comunicación por paso de mensajes. Además, se explica la utilidad de las redes de Petri para reproducir trazas grabadas previamente. Se concluye con los requisitos que debe tener un juego para poder utilizar el nuevo componente de grabación de trazas de forma correcta y sin aplicar cambios significativos.

4.1. Introducción

Como se ha mencionado anteriormente en el capítulo de testing, la aparición de videojuegos complejos requiere una nueva forma de realizar pruebas que permita demostrar la corrección de todos los aspectos del juego. Puesto que los test de unidad no son una solución completa debido a que no pueden probar absolutamente todo lo que va a pasar en un videojuego cuando un usuario real lo utilice, introducimos la grabación y reproducción de trazas como alternativa a los test de unidad. De este modo, nuestros test pasarán a probar también el diseño del videojuego y la jugabilidad del mismo, que están más próximos a la experiencia del jugador. Gracias a la grabación y

reproducción de trazas, se podrán generar test que comprueben no sólo el software, sino también realicen evaluaciones del mapa y la distribución de las entidades en él, entre otras cosas. Así, no será necesario disponer de beta tester humanos que jueguen una y otra vez a los videojuegos para comprobar que los niveles son correctos. Además, el uso de trazas grabadas previamente permite facilitar la tarea de reproducir errores que serían muy difíciles de encontrar si los testeadores tuvieran que repetir sus acciones innumerables veces.

Para poder diseñar test que se puedan usar con ese propósito, en primer lugar se deben grabar trazas de partidas del videojuego. Se generarán ficheros de datos que contendrán la información de las acciones llevadas a cabo por el usuario. Para ello se crearán dos ficheros distintos: uno con los movimientos a bajo nivel y otro con las acciones a más alto nivel. El primero de los ficheros contendrá cada una de las pulsaciones de teclas que lleva a cabo el usuario y el tick en el cual se ha llevado a cabo, sin más información que esa. En cambio, el segundo fichero se generará a partir de los mensajes que se envían los componentes, por lo que será posible conocer las acciones a más alto nivel que se han producido en el juego.

Por último, este capítulo cuenta cómo se usan los ficheros anteriores para reproducir trazas de juegos, ya que esa reproducción será la que dará lugar a los test creados específicamente para probar juegos. Para ello se utiliza el modelo de las redes de Petri, que permitirá que sea más fácil reproducir las trazas en orden de la forma más adecuada, facilitando que la reproducción sea correcta y el test pueda ser superado.

4.2. Grabación de trazas del videojuego

Existen dos formas distintas de grabar ejecuciones de un juego para reproducirlo posteriormente. La primera de ellas es la entrada directa del usuario (`User Input`), que permite guardar en un archivo de forma ordenada todas las teclas pulsadas y movimientos del ratón realizados por el usuario. La otra forma de grabar trazas consiste en guardar acciones a alto nivel, que es posible gracias a la introducción de un nuevo componente en el juego. A continuación se explica detalladamente en qué consiste y cómo se ha implementado cada una de estas alternativas.

4.2.1. Grabación mediante entrada directa del usuario

Denominamos entrada directa del usuario a todas aquellas teclas que éste pulsa en su teclado o a los movimientos realizados con el ratón para llevar a cabo distintas acciones en el juego. Estas pulsaciones se guardarán en un fichero de texto, en el cual se especificará la tecla pulsada asociada al momento del tiempo en el cual se ha pulsado. El objetivo de grabar directamente

esa información es disponer de un fichero en el cual conste absolutamente todo aquello que el usuario ha hecho en el juego para poder reproducirlo posteriormente sin modificación alguna.

La utilidad de dicho fichero se pone de manifiesto cuando queremos llevar a cabo distintos tipos de pruebas en las cuales sabemos con certeza que conseguir los objetivos o llegar al final del nivel se va a poder hacer de la misma manera que cuando el usuario jugó inicialmente. Ejemplos de pruebas en las cuales puede resultar interesante esto son las pruebas de compatibilidad, las pruebas de regresión o las pruebas de integración.

Las pruebas de compatibilidad consisten en llevar a cabo los mismos test en varias ocasiones para garantizar la compatibilidad del juego en distintas plataformas de hardware o en distintos sistemas operativos. Así, poder realizar con exactitud los mismos movimientos o acciones se convierte en una tarea muy importante que ayudará a comprobar que el juego se comporta como esperamos. Por otro lado, las pruebas de regresión buscan errores en el comportamiento del juego cuando se han realizado cambios de software que no se pueden apreciar a simple vista en el juego (mejoras, refactorización del código o rediseño del juego). Al no haberse modificado el juego de forma directa, reproducir los mismos test permitirá asegurar que los cambios introducidos no han roto el sistema. Por último, las pruebas de integración intentarán descubrir fallos al combinar varios módulos que ya habían sido probados en uno solo.

Por último, es necesario mencionar que la información que contiene el fichero que se genera al grabar la entrada directa del usuario es de bajo nivel. Por un lado contiene el valor exacto de cada tecla que el usuario pulsa en el teclado, y por el otro contiene la información de cada movimiento y click del ratón, que normalmente supondrá una reorientación de la cámara en el juego. Todo lo anterior llevará asociado el tiempo en el que se ha producido la pulsación o el movimiento, ya que posteriormente usaremos esos datos para reproducirlos exactamente en el mismo instante de tiempo.

4.2.2. Grabación mediante nuevo componente: CRecorder

Como se ha dicho en el capítulo anterior, al utilizar una arquitectura basada en componentes e implementada con un mecanismo de paso de mensajes, es muy fácil introducir un nuevo componente en el juego que responda de la forma que nosotros queremos a los distintos mensajes que se van generando.

De forma general, se ha creado un nuevo componente, al cual se ha denominado `CRecorder`, cuyo objetivo es registrarse como listener de las entidades que nos interesan y posteriomente generar un archivo (*log*) con los resultados obtenidos. Existirá un fichero de texto (normalmente denominado "blueprints") que contendrá la lista de entidades del juego con todos los

componentes asociados a cada una de esas entidades. Para registrar nuestro componente como listener de una de las entidades, lo único que habrá que hacer será incluir el nombre del componente (`CRecorder`) en la lista de componentes de esa entidad.

Gracias al paso de mensajes, cuando se produzca un evento interesante en el juego se enviará un mensaje a todas las entidades. Éstas enviarán el mensaje en cuestión a todos los componentes que estén registrados como listeners de la misma, y ellos decidirán si quieren tratar el mensaje y hacer algo en respuesta. A continuación se habla de los detalles concretos de la implementación de estos conceptos en Time and Space, ya que a pesar de ser específicos de este juego (véase 5) ayudan a realizar la exposición con claridad.

En nuestro caso, se han creado filtros de mensajes en el juego, y cada uno de ellos trata varios de los mensajes que recibe. Para elegir qué mensajes trata y cuáles no, se ha creado una función *accept* en cada filtro, que devuelve *True* si quiere tratar el mensaje, y *False* en caso contrario. Por ejemplo, se ha implementado un filtro denominado `PlayerFilter`, que escribe en un log la información importante de todas las acciones realizadas por el usuario. Otro de los filtros que se han implementado se llama `SwitchFilter`, que se ocupa de escribir los mensajes de *TOUCHED, UNTOUCHED* y *SWITCH* que se generan en la aplicación cuando alguien pulsa un botón. Además, cuando se crean los filtros se puede especificar cómo se denominará el log en el cual cada uno de ellos va a escribir. Para ello se ha definido una interfaz `ILogger`, que permite implementar distintos Loggers de forma que cada uno de ellos escriba los mensajes de forma distinta. Así, podríamos tener distintos ficheros generados, cada uno de ellos con un contenido distinto (tanto en formato como en contenido), lo cual resultará útil a la hora de reproducir las trazas.

El `CRecorder` no sólo será útil para guardar la información de los eventos interesantes que han ocurrido relacionados con el jugador. Al tratarse de un componente, podríamos incluso crear un fichero que registre cambios en la física del mapa, en los sonidos o en otra serie de cosas que son independientes del juego en sí mismo.

Las trazas que guarda el `CRecorder` registran acciones a alto nivel. Gracias a él, podemos conocer eventos como la activación de un botón, el disparo de un enemigo, la activación de una plataforma o el movimiento de una puerta. Sin embargo, no existe una forma directa de reproducir esas acciones, ya que sólo se guardan el nombre de la acción, el momento en el cuál se ha llevado a cabo y las entidades asociadas a ella. En el caso de la pulsación de un botón, las entidades asociadas serán el jugador que lleva a cabo la acción y el botón que se ha pulsado. En las siguientes secciones se realizará un estudio sobre cómo hemos realizado la adaptación y reproducción de trazas del juego para poder llevar a cabo test que vayan más allá de los test de

unidad. Guardar en el fichero las acciones a alto nivel que se van registrando durante la ejecución proporcionará una forma inteligente de reproducir juegos guardados anteriormente. Esta propuesta mejora considerablemente los resultados que se pueden obtener al realizar test de unidad en juegos, ya que permitirá ahorrar en costes de pruebas y evitar trabajo a los beta testers debido a la posibilidad de reproducir las acciones en mapas que hayan sufrido modificaciones.

4.3. Adaptación y reproducción de trazas del juego

Como se ha dicho anteriormente, la posibilidad de reproducción de trazas grabadas se puede aprovechar para mejorar los resultados que se consiguen al usar test de unidad. Los test de unidad tienen una utilidad bastante limitada en cuanto a juegos se refiere, ya que todos los problemas que pueden surgir debido a la imprevisibilidad del usuario son muy difíciles de probar en la etapa de test automáticos. Otra de las limitaciones surge al desarrollar un juego en el cual la cantidad de jugadores resulta ser masiva, como pueden ser los juegos en línea.

Los test basados en la reproducción de trazas van más allá de los test de unidad. Los últimos comprueban que ciertas secciones de código responden como deben, pero en ningún caso podrán probar que la jugabilidad del juego sigue siendo óptima. Los juegos, después de sufrir modificaciones, deben seguir siendo completos, entretenidos y con un nivel de dificultad que permita al jugador disfrutar de la partida. Reproducir trazas permitirá tener un banco de pruebas almacenado que facilitará la tarea de probar que todo funciona como se espera incluso después de haber cambiado los mapas del juego. Esto último es importante, ya que los diseñadores de juegos pueden modificar un mapa gracias a que los distintos niveles y entidades están definidos en un fichero de texto que es completamente independiente del código. De esta forma, cuando se quieran introducir cambios, no será necesario involucrar al programador del juego para nada, pudiendo añadir esas modificaciones y después reproducir las trazas grabadas previamente para comprobar que todo funciona.

Para ilustrar lo anterior, tomemos el ejemplo de los juegos en línea con una cantidad muy elevada de jugadores. En una primera aproximación, se puede intentar realizar una prueba con muchos jugadores que nos permita ver si todo es correcto y funciona de acuerdo a lo esperado. En esta ejecución del juego, se grabará todo lo que está haciendo cada uno de los jugadores en el juego, y también cómo lo está haciendo. Las ventajas de grabar esa información son varias: cuando surja un error o un bug en el juego que impida continuar con la prueba, se tendrá la información exacta de lo que ha ocurrido para llegar a esa situación, lo cual puede ayudar a los programadores y diseñadores a corregir el error. Posteriormente, cuando se considere que

ese error ya está corregido, en lugar de tener que volver a usar innumerables personas para que prueben de nuevo el juego se pueden ejecutar las mismas trazas grabadas en la primera ocasión. De esta forma, no será necesario volver a jugar para probar que ese fallo está solventado. Además, se evita que en una nueva prueba del juego las acciones que llevan a cabo los usuarios sean completamente distintas y no pongan de manifiesto el error, ya que se podrá reproducir exactamente igual que en la primera ocasión.

4.3.1. Reproducción directa de trazas del juego

La reproducción directa de trazas tiene como objetivo la realización de pruebas muy concretas, ya que no siempre que queramos realizar test será porque se hayan introducido cambios en el videojuego. Uno de los ejemplos más significativos de esto es llevar a cabo las pruebas en distintos dispositivos para comprobar que en todos se obtiene siempre el mismo resultado y que la jugabilidad no se ve afectada por el cambio de hardware (pruebas de compatibilidad). Más ejemplos de la utilidad de la reproducción directa son las pruebas de regresión, de escalabilidad o de integración, en las cuales se puede probar que el videojuego sigue funcionando a pesar de haber introducido cambios.

Para que sea posible reproducir directamente las trazas del juego, en primer lugar es necesario cargar el fichero que se había generado (a bajo nivel) y guardar esa información en una cola de acciones que se reproducirán cuando el tick del juego coincida con el tick que había guardado en el fichero de texto. En esta alternativa no es posible adaptar las trazas, sino que se reproduce exactamente cada pulsación que llevó a cabo el usuario en la primera ejecución del juego.

Es necesario crear nuevas funciones en el gestor de entrada del usuario para poder inyectar directamente las teclas y movimientos en el juego. Para ello se han creado ciertas funciones que simulan el uso del teclado y del ratón enviando el nuevo estado a los listeners que estén registrados como listeners del teclado y del ratón, respectivamente. Están implementadas de forma que se pueda llamar a estas funciones desde la clase que comprueba si hay que inyectar en el tick actual. La implementación concreta de estas funciones aplicada al videojuego descrito en el capítulo 5 puede verse en el apéndice A.1. Estas funciones han sido añadidas al InputManager existente en el proyecto GUI.

4.3.2. Reproducción de acciones del usuario. Redes de Petri

El objetivo de grabar sesiones de un juego es evitar jugar múltiples veces si los cambios introducidos en el juego no son significativos. En la alternativa de reproducción de las acciones del usuario, vamos a cargar en el juego un fichero que contenga todas las acciones a alto nivel que inicialmente llevaron

a cabo los beta tester. Estas acciones, como se ha dicho anteriormente, se usarán para reproducir el juego nuevamente en un entorno de test. Esto contribuye al desarrollo de test que sean repetibles y se puedan llevar a cabo de forma automática.

Algunos de los atributos generales de los juegos son las acciones que pueden ocurrir en el juego (las que el jugador puede realizar o las que llevan a cabo las diferentes entidades), el mapa físico o el estado del juego. Cuando el objetivo que se persigue al grabar una sesión es reproducirla posteriormente, es necesario pensar qué atributos tendremos que guardar en el archivo de log para que después se pueda reproducir sin problemas. Por ejemplo, imaginemos una acción en la cual el jugador coja un arma del juego. Para reproducir esa acción, es necesario saber qué jugador (o entidad) lleva a cabo la acción, qué acción está ocurriendo y las entidades asociadas a esa acción. Consideramos entidades asociadas tanto al jugador que lo lleva a cabo como la entidad sobre la que se lleva a cabo, en este caso el arma.

Un jugador no podrá coger un arma si no se encuentra en el mapa o si el jugador no se encuentra lo suficientemente próximo a ella como para cogerla. Por tanto, el estado del juego debe ser lo más parecido posible al que tenemos almacenado en el log cuando el tiempo se vaya acercando al momento en el cual la acción ocurrió inicialmente. Cuanto más parecido sea, más fácil será que replicar las acciones sea factible. Es necesario destacar que no es necesario guardar la posición del arma que tiene que coger el jugador, ya que esta información está almacenada en un fichero distinto que se carga en el juego y cuya información es accesible en tiempo de ejecución.

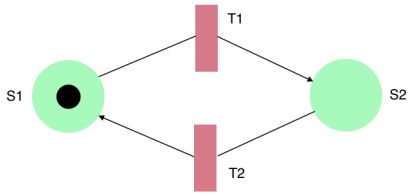
Con el propósito de modelar todos estos atributos que son necesarios para reproducir el juego, utilizaremos unas representaciones muy potentes denominadas redes de Petri temporizadas, que nos van a resultar muy útiles para reproducir juegos.

4.3.2.1. Modelando el juego con redes de Petri temporizadas

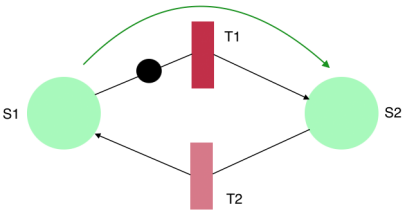
Las redes de Petri son un lenguaje de modelado que se pueden describir tanto gráficamente como matemáticamente. La descripción gráfica se representa como un grafo dirigido compuesto por nodos, barras o cuadrados, arcos y tokens. Los elementos de un modelo de una red de Petri son los que siguen:

- **Lugares (places)**: Se simbolizan mediante nodos. Los lugares son elementos pasivos de una red de Petri, y representan condiciones.

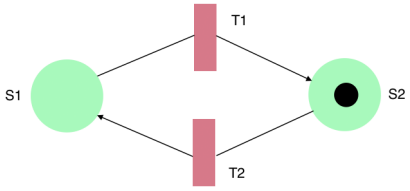
- **Transiciones**: Las barras o cuadrados representan las transiciones de la red, que son las acciones o eventos que pueden causar que se produzca un cambio de lugar en una red de Petri. Por tanto, son elementos activos. Las transiciones podrán usarse (ser activadas) si hay suficientes marcas que consumir cuando la transición se active.



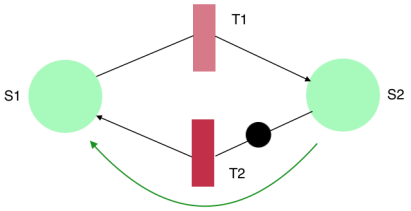
(a) Estado 1: Puerta cerrada



(b) Transición 1: El jugador abre una puerta



(c) Estado 2: Puerta abierta



(d) Transición 2: El jugador cierra la puerta

Figura 4.1: Modelado de las acciones de abrir y cerrar una puerta usando una red de Petri.

- **Marcas (tokens)**: Cada uno de los elementos que pueden activar una transición se denomina marcas o tokens. Junto con los lugares, modelan los estados del sistema. Cada vez que una transición se activa, una marca se mueve de un lugar a otro, haciendo que el estado de la red de Petri cambie. Las marcas se representan mediante un punto en el interior del nodo que representa cada lugar.

- **Arcos**: Los lugares y las transiciones están conectados por arcos. Un arco puede conectar un lugar a una transición o en el sentido contrario, pero nunca podrán conectar dos lugares o transiciones entre ellos.

La figura 4.1 muestra un ejemplo de modelado de las acciones de abrir y cerrar una puerta usando una red de Petri. 4.1a muestra el estado inicial, en el cual la puerta está cerrada y un token está presente en ese estado. 4.1b muestra la transición que tiene lugar cuando un jugador abre la puerta. Después, el token se mueve de S1 a S2, haciendo que el nuevo estado pase a ser 4.1c. Si el jugador vuelve en ese momento a llevar a cabo la acción de cerrar la puerta (mostrada en la transición 4.1d), el token volverá al estado inicial de nuevo 4.1a. Cabe destacar que en las figuras 4.1b y 4.1d los tokens están en medio de uno de los arcos. Sin embargo, los modelos de las redes de Petri no permiten que los tokens estén fuera de los lugares. En este ejemplo se muestra así para enfatizar el movimiento del token, que pasa de un estado

al otro cuando se activa la transición.

Las redes de Petri clásicas pueden extenderse para asociar un tiempo a cada transición. Cuando las transiciones duran más que una unidad de tiempo cada una, se denomina a estos nuevos modelos redes de Petri temporizadas. Introducir el tiempo en el modelo es esencial para reproducir juegos, ya que las acciones no son inmediatas normalmente. Por ejemplo, si queremos reproducir una acción como "abrir una puerta", en primer lugar el jugador deberá encontrarse físicamente al lado de la puerta y después llevar a cabo la acción de abrirla. Esto significa que cada acción puede ocupar más de una unidad de tiempo, y otras unidades podrían incluso estar llevándose a cabo a la vez. Por ese motivo, modelar los juegos como redes de Petri temporizadas hacen que el resultado sea más sencillo que el modelado como una máquina de estados.

Después de cargar un fichero de trazas con el propósito de reproducirlo de nuevo, las acciones tienen que llevarse a cabo en un estado lo más parecido posible al estado original. Es más, las acciones están ordenadas casi siempre: un jugador no puede andar a través de una puerta si la puerta no se ha abierto previamente. Las acciones que se realizan pueden afectar a varias entidades, no sólo a una, por lo que usar redes de Petri para modelar el comportamiento de un juego parece razonable.

Cuando detectamos que un jugador ha realizado una acción en la primera ejecución del juego y es posible activar la transición de la red de Petri, los mensajes que correspondan a esa acción deben generarse e introducirse en la aplicación. Al hacer eso, no existirá ninguna diferencia entre los mensajes que se generan cuando una persona real está jugando y los mensajes que se inyectan de forma simulada. Los componentes aceptarán esos mensajes en ambas versiones, los procesarán y responderán de acuerdo al contenido. Por esa razón, el fichero resultante debe ser exactamente el mismo y de esa forma será posible ver que las acciones que han ocurrido en el juego son las mismas y no han cambiado.

4.4. Requisitos para la reproducción de trazas

En las secciones anteriores se ha hablado de la reproducción de partidas de juegos como forma de llevar a cabo pruebas que van más allá de los conocidos test de unidad que se usan tradicionalmente.

Debido a que la reproducción de acciones a alto nivel necesita utilizar una IA para implementar la navegabilidad del jugador y que sea posible encontrar los destinos y desplazarse hacia ellos, es necesario que esta IA sea lo suficientemente precisa como para poder llevar a cabo esas acciones. En general, se puede utilizar la IA implementada para los enemigos (non-player characters) para llevar a cabo la reproducción de trazas y mover al jugador. Sin embargo, si la IA de los NPCs no es lo suficientemente buena,

será necesario introducir una IA específica para la reproducción de trazas o ampliar la que ya existe para que esa reproducción sea viable.

Como ejemplo del mal funcionamiento de la reproducción de trazas en caso de que la IA del juego no esté lo suficientemente avanzada tenemos el que sigue. Imaginemos que al jugar, necesitamos subirnos a una plataforma, saltar para llegar a un nivel distinto del que nos encontrábamos inicialmente y posteriormente disparar a un enemigo que se encuentra en el nivel inferior. En este caso, a la hora de realizar la reproducción es necesario que la IA sea capaz de encontrar caminos entre distintos niveles, ya que tendrá que poder subir la plataforma sin ayuda del jugador humano. Además, para poder disparar al enemigo, es necesario que pueda detectar dónde está este, ya que si no será imposible dispararle.

El ejemplo anterior pone de manifiesto la necesidad de que la IA del juego esté lo suficientemente bien implementada y sea capaz de reaccionar a diversas situaciones. Las búsquedas de caminos deben ser avanzadas para que el resultado sea óptimo. Si esto no fuera así, podría darse el caso de que el jugador se quede parado en medio de la reproducción porque no sepa como reaccionar a la traza actual.

En el capítulo 5 se verá un ejemplo de videojuego en el cual la reproducción no funciona cuando los objetivos están en niveles (o altura) distintos, ya que la IA sólo dispone de búsqueda de caminos que se encuentran a la misma altura. Además, será necesario introducir *waypoints* en el mapa para que el jugador sea capaz de llegar a sitios que están fuera de su campo visual. Los waypoints son puntos distribuidos por el mapa que indicarán al jugador por dónde tiene que ir en caso de que falle la búsqueda directa del camino por no tener el objetivo a la vista.

Capítulo 5

Time and Space

There is a language that
extends beyond words.
Paulo Coelho

Resumen: En este capítulo se cuenta cómo se ha introducido y aplicado lo que se ha contado previamente en el capítulo de grabación y reproducción de trazas a un videojuego en particular. Comenzamos introduciendo Time and Space y definiendo el dominio del juego para posteriormente explicar el estilo de las trazas generadas y cómo se utilizan para llevar a cabo la reproducción del juego.

5.1. Introducción

En este capítulo se presenta Time and Space (Blázquez Checa et al., 2013-2014) y se define un estándar de representación de las trazas que se van a generar cuando el humano experto juega al mismo. Para ello se planteará qué es importante guardar, cómo hay que guardarlo y qué se necesita para poder reproducir una traza posteriormente.

En la primera sección se describe el juego, en la segunda sección el dominio de éste, y en la última se proporciona un ejemplo de las trazas que se van a generar cuando la opción de grabación de trazas esté activada.

5.2. Time and Space

Para la realización de este trabajo no se ha implementado un videojuego nuevo de cero, sino que se ha reutilizado uno creado por los alumnos del Máster de Videojuegos de la Universidad Complutense de Madrid. Se ha

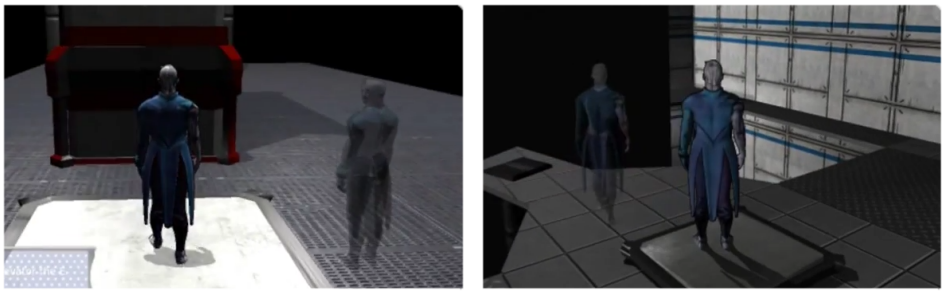


Figura 5.1: Capturas de pantalla de Time and Space

elegido este juego por la aparición de dos elementos novedosos: las copias del jugador y el uso del tiempo para que estas copias puedan reproducir las acciones previas de éste.

5.2.1. Objetivos y niveles

Se trata de un juego de niveles cortos en el cual el objetivo es llegar al portal de luz que se encuentra al final de cada nivel. Para ello, el jugador deberá ayudarse de copias de sí mismo que realizan exactamente los mismos movimientos que en su ejecución anterior. Por ejemplo, para que el jugador sea capaz de cruzar una puerta una copia anterior debe estar situada encima de un botón. De lo contrario, la puerta se cierra y el jugador no será capaz de pasar. Otro ejemplo de ello es usar la creación de copias para formar una barrera entre el jugador y un enemigo. De esta forma, los ataques los recibirán las copias y el jugador será capaz de llegar al objetivo final del nivel. Se pueden ver dos capturas de pantalla del juego en la figura 5.1

Time and Space es un videojuego implementado utilizando una arquitectura basada en componentes que utiliza paso de mensajes. Al jugar, los mensajes generados en la aplicación se almacenan en una cola de mensajes. Cuando posteriormente el jugador crea una copia (o clon), se utilizan los mensajes guardados para repetir las acciones de esas copias, que reproducen exactamente lo mismo que cuando el jugador principal lo llevó a cabo.

5.2.2. Acciones y mensajes especiales

Existen ciertas acciones que los clones nunca tendrán que reproducir y que por tanto no se guardarán. Ejemplo de esto es la propia creación de copias, ya que éstas no mueren a no ser que sean atacadas por un enemigo. Después de reproducir la lista de movimientos completa que tenían guardada, las copias mantienen la última posición en la que se encontraban antes de clonarse, de modo que si estaban posicionadas encima de un interruptor que mantenía abierta una puerta esta no se va a cerrar.

Otro tipo de acciones especiales en la aplicación son los de "reiniciar nivel sin clonarse". El jugador puede elegir desechar las últimas acciones realizadas desde la creación de la última copia si no está conforme con los movimientos que ha ejecutado. El mensaje generado a partir de esta acción es otro ejemplo de los mensajes que son exclusivos del jugador principal y que no se guardan para que posteriormente lo ejecuten los clones.

5.3. Dominio del juego

Existen cuatro tipos de elementos en el juego: Entidades, acciones, eventos y objetivos. Las siguientes secciones explican qué se entiende por cada elemento.

5.3.1. Entidades

Las entidades son aquellos elementos del juego que pueden interactuar de una forma concreta con el resto del mapa, pero también aquellos con los que se puede interactuar. El jugador y sus copias son entidades que pueden realizar ciertas acciones que modificarán el estado del juego. Por otro lado, existen entidades como interruptores o botones que pueden ser accionados por el jugador. Las entidades que se han identificado son las siguientes: el jugador, copias del jugador, un switch, un arma, una torreta, un botón, una plataforma, un puente, un enemigo o incluso el tiempo.

5.3.2. Acciones

Las acciones son operadores que pueden aplicarse al juego. Puede implicar que ciertas entidades interactúen con otras. Cada acción tiene unos ciertos parámetros: Quién la lleva a cabo, sobre quién actúa, qué elementos sufren cambios después de llevarse a cabo dicha acción, etc.

Para saber cuándo se pueden llevar a cabo las acciones, se pueden especificar ciertas condiciones: precondiciones, postcondiciones y condiciones de fallo. Por ejemplo, para pulsar un switch asociado a una puerta, la precondición es estar situado sobre el switch (tener las mismas coordenadas). La postcondición asociada es el movimiento de la puerta (en este caso, apertura).

- **Movimiento del avatar:** andar, saltar, correr, agacharse o girar son algunas de las acciones relacionadas con los movimientos del avatar. También las acciones que indican que se dejan de hacer los movimientos anteriores.
- **Acciones del jugador:** en este grupo se incluyen acciones como los disparos, los giros, hacer zoom de la cámara para apuntar, cambiar de arma, pulsar un botón, dejar de pulsar un botón, etc.

- **Acciones especiales:** como se ha dicho anteriormente, algunas de las acciones que realiza el jugador no tienen que ser reproducidas por sus copias. En ellas se incluyen los "poderes especiales" del jugador: la creación de copias, poner el tiempo en reproducción turbo, resetear el nivel sin crear copia o coger un objeto.

Todas las acciones mencionadas anteriormente serán las que nos interesará guardar en el fichero de texto para más tarde poder reproducirlas.

5.3.3. Eventos

Los eventos son típicamente sucesos del juego que se desencadenan al realizarse una acción. Dicho de otra forma, las postcondiciones de las acciones se convertirán en eventos. A pesar de ello, hay que destacar que postcondiciones y eventos son cosas distintas, ya que los eventos nos van a ayudar a reconocer qué objetivos persigue el jugador.

A la hora de reconocer objetivos, es importante saber si el jugador o una de sus copias ha accionado un switch, pero lo realmente valioso es el evento desencadenado por esa acción, que es independiente de quién la realiza.

5.3.4. Objetivos

En Time and Space hay varios niveles. Existe un objetivo final en cada nivel, que consiste en atravesar el portal de luz ("meta"). Además, para cada nivel se pueden definir subobjetivos, que son pequeñas metas que ayudan al jugador a avanzar de modo que la meta final esté más próxima o más accesible. Por ejemplo, si el portal de luz está detrás de una puerta, pulsar el switch que abre la puerta se convierte en un subobjetivo necesario para llegar al final de ese nivel.

Los subobjetivos serán muy importantes a la hora de reproducir las acciones del juego, ya que habrá que intentar que se cumplan de forma ordenada. No se podrá pasar al siguiente objetivo hasta que el anterior haya ocurrido, lo que implica que podremos aplicar las redes de Petri temporizadas para comprobar las precondiciones y los tiempos de cada acción y así ver si podemos reproducir el siguiente o no.

5.3.5. Otras condiciones

Se ha considerado la existencia de otro tipo de condiciones (por ejemplo, condiciones de éxito). Sin embargo, se ha llegado a la conclusión de que en este juego (Time and Space) no son necesarias, ya que el resultado o las consecuencias de todas las acciones es inmediato. Existen otro tipo de juegos en los cuales las acciones pueden fallar por motivos ajenos al jugador. Es el caso de los juegos en los que se pueden realizar construcciones, en los

que empezar a construir una unidad no implica que la unidad se construya con éxito, ya que puede ocurrir que el enemigo la destruya antes de que esté completamente acabada. En ese caso sí que tendría sentido añadir más condiciones. Un ejemplo de esto sería el juego Towers, véase (Gómez-Martín et al., 2009).

5.4. Grabación de trazas en Time and Space

En Time and Space se ha introducido un archivo de configuración que permite indicar desde fuera si se quiere activar la grabación o la reproducción de trazas. Una vez que la grabación de trazas está activada, el testeador puede jugar la partida mientras se van generando los dos archivos: el archivo a bajo nivel con las teclas pulsadas y el archivo a más alto nivel con las acciones del usuario.

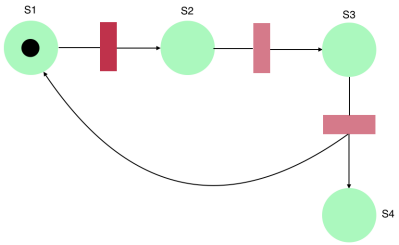
Al activar la grabación, toda la información interesante que ocurra en el juego se estará almacenando en los archivos. Para decidir qué acciones son interesantes y cuáles no, y también el formato en el cuál se van a escribir hemos implementado *filtros* y *loggers*. Los filtros serán los encargados de capturar ciertos tipos de mensaje, mientras que los loggers se encargarán de escribir los mensajes capturados por los filtros.

Se han implementado distintos filtros, y cada uno de ellos se encarga de aceptar diferentes mensajes. La existencia de los filtros es necesaria, ya que se genera una cantidad muy elevada de eventos en el juego, por lo que se vuelve imposible tratarlos todos según se crean. Además, cada mensaje consta de argumentos que en general nada tienen que ver los unos con los otros. Es por esto que cada filtro tratará un mensaje o unos cuantos mensajes que contengan información similar y que puedan englobarse en uno solo. Los filtros tienen asociado un logger. Esto permite que dependiendo del mensaje que estemos guardando la información pueda estar en un formato o en otro completamente distinto. Además, cada logger puede tener asociado un fichero distinto, lo que permitiría que al reproducir trazas se pudiera leer la información en formatos diferentes.

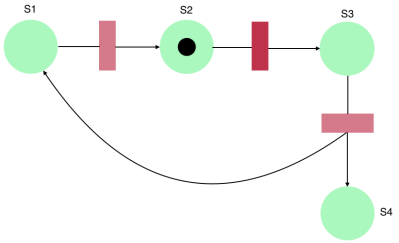
En el apéndice A.2 se puede ver un ejemplo de logger, mientras que en A.3 se puede consultar uno de los filtros implementados, el *SpecialActionFilter*.

5.5. Reproducción de partidas a partir de trazas grabadas

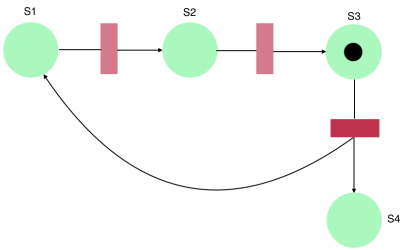
A pesar de que la reproducción de partidas tiene como objetivo llevar a cabo diferentes test, antes de realizar la implementación del sistema de testing se han realizado numerosas pruebas para comprobar que la reproducción se hacía correctamente.



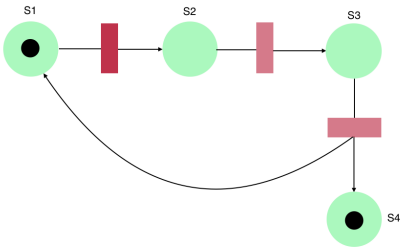
(a) Estado 1: Un sólo jugador. Puerta cerrada.



(b) Estado 2: El jugador ha pulsado un botón y la puerta está abierta.



(c) Estado 3: El jugador ha modificado la velocidad del juego.



(d) Estado 4: El jugador se ha clonado, luego vuelve al principio y existe una copia del mismo.

Figura 5.2: Modelado de apertura de puerta y copia por el jugador.

Para reproducir trazas se han modelado las acciones que hay que llevar a cabo como una red de Petri temporizada. Esto quiere decir que una vez que tenemos un fichero con diferentes acciones que el usuario ha llevado a cabo, estas se van a reproducir en orden siempre que sea posible. No se intentará reproducir la siguiente acción disponible en el fichero a no ser que todas las acciones previas se hayan llevado a cabo correctamente. Sin embargo, si los cambios introducidos en el mapa son significativos, la reproducción fallará y no será posible llevar a cabo los objetivos.

Las redes de Petri, debido a la existencia de copias, tendrán una estructura de retroalimentación, ya que las copias repetirán las acciones cuando el jugador se clone. Esto se puede ver en la figura 5.2, donde se ha modelado el principio del nivel 1 de Time and Space. Además, la distribución de los objetos en el nivel 1 puede verse en la figura 5.3.

En la figura 5.2a se muestra el primer estado del juego, en el cual sólo tenemos un primer jugador y una puerta cerrada. Además, se supone que tenemos la posibilidad de pulsar un botón para que la puerta se abra, por lo que la transición del estado S1 al estado S2 está activa, mientras que las demás no lo están. Cuando el jugador pulsa el botón para abrir la puerta, el token pasa al estado S2 y tenemos lo que se ve en 5.2b. En este caso, el

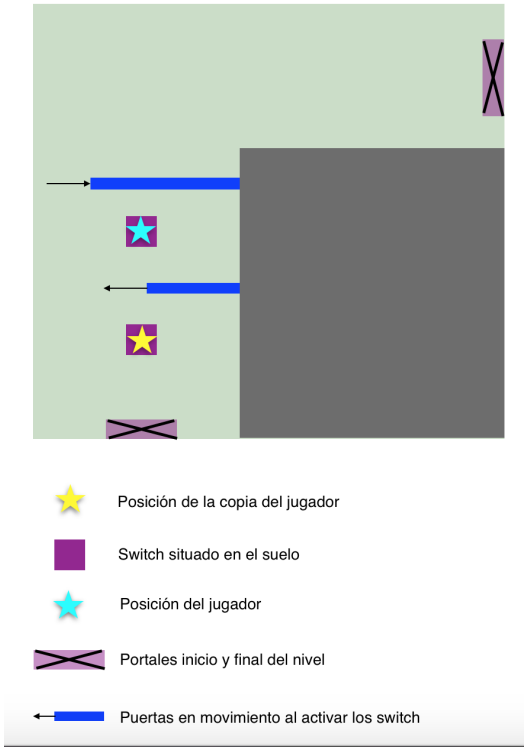


Figura 5.3: Esquema del nivel 1 de Time and Space

jugador está encima del botón y la puerta se mantiene abierta. Supongamos que se puede realizar la acción de modificar la velocidad del juego. En este caso, si el jugador pulsa la tecla que modifica la velocidad el token del jugador pasará al estado S3, representado en 5.2c. Cuando pase el tiempo suficiente para que en la siguiente reproducción el nuevo jugador pueda pasar por la puerta mientras la copia pulsa el botón, el jugador se clonará, pasando al estar en el estado S1 de nuevo. Esto se puede ver en la figura 5.2d. Sin embargo, se habrá creado un nuevo token, correspondiente a la copia que se acaba de crear del jugador, que se encuentra en el estado S4. Esto es sólo una pequeña muestra de la modelización completa del nivel 1.

En la figura 5.3 podemos ver la distribución de las entidades en el nivel 1. El esquema se ha realizado en un momento en el cual existe una copia del jugador y está a punto de clonarse para conseguir una segunda copia que le ayude a pasar por la segunda de las puertas.

Las redes de Petri utilizadas tienen que ser temporizadas ya que por ejemplo, cuando la copia abre la puerta, el jugador nuevo tendrá que esperar a que esta acabe de abrirse para poder pasar a través de ella y continuar con la reproducción del juego. Es por ello que las transiciones de la red de Petri no son inmediatas, sino que llevan asociado un tiempo que es dependiente del

caso que estamos tratando. Por ello, en la implementación de la reproducción de trazas tenemos que reflejar esta situación para que las acciones no se encadenen unas detrás de otras sin ningún sentido. Así, teniendo en cuenta el momento del juego en el que se tiene que llevar a cabo cada acción y los objetivos que tienen que haberse cumplido antes de lanzarla, podemos realizar una reproducción exacta del juego que después podrá utilizarse para diseñar y pasar test de unidad específicos para este videojuego.

5.5.1. Algunas pruebas realizadas con Time and Space

Se han llevado a cabo diferentes pruebas en distintos niveles del juego para comprobar que la grabación y la reproducción de trazas funciona de forma adecuada. Para ello, se han grabado ejecuciones de trazas de diferentes niveles y se han reproducido después, modificando el mapa para comprobar si los resultados que se obtienen son los esperados.

Algunos de los test que se han hecho han sido los siguientes:

- En primer lugar se ha llevado a cabo la grabación de trazas del nivel 1 del juego. Para realizar los test, inicialmente se ha cambiado el mapa moviendo los interruptores a sitios que sean alcanzables desde la primera posición que inicialmente tenían. Después se ha llevado a cabo el mismo test, pero cambiando también la marca de fin de nivel (el arco de luz por el cual tiene que pasar el jugador). Ambos test se han completado satisfactoriamente después de llevar a cabo los cambios indicados, y los objetivos del nivel se han concluido con éxito, pudiendo llegar al final del nivel sin problema. En los dos casos, la reproducción de trazas tiene éxito y el test se pasa satisfactoriamente, demostrando así que la reproducción es estable a cambios en el mapa. Otro de los test que hemos hecho ha consistido en poner uno de los interruptores que el jugador tenía que pulsar detrás de una puerta cerrada. En este caso, podemos ver cómo el jugador detecta que la nueva posición del interruptor no es alcanzable y no modifica su posición, quedándose en el mismo lugar que tenía antes de detectar que tenía que pulsarlo. Por tanto, el test falla, lo cual prueba que el cambio introducido por el diseñador en el mapa ha roto el nivel. El ejemplo de la primera de estas pruebas puede verse en `https://www.youtube.com/watch?v=kzCBL7mZois`.

- También se han grabado trazas del nivel 4, un nivel en el cual el jugador necesita tres copias diferentes de sí mismo para poder superar el nivel. El esquema del nivel puede verse en la figura 5.4. Para poder llegar al final del nivel, el jugador tiene que pasar por una plataforma en movimiento. Además, también es necesario que desactive los rayos que hay justo delante de la marca de fin de nivel, ya que si intenta pasar por ellos sin desactivarlos morirá. Es por esto que el jugador necesita clones de sí mismo para conseguir llegar al final. Para llevar a cabo los test,

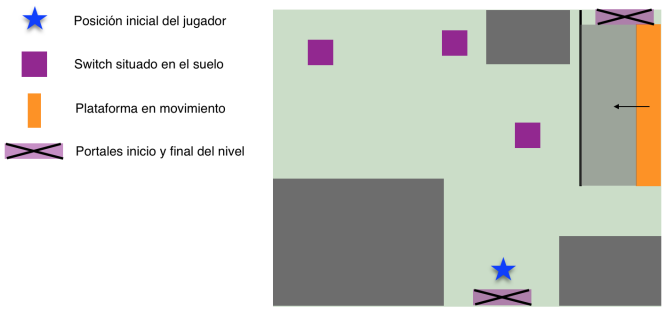


Figura 5.4: Esquema del nivel 4 de Time and Space

se han intercambiado los interruptores entre ellos. Cada interruptor realiza una acción necesaria para llegar al final. Al intercambiarlos, habremos colocado los interruptores en posiciones más cercanas o más lejanas al jugador. Al lanzar los test podemos ver que a pesar de todos los cambios, sigue siendo posible completar el nivel sin dificultad y el test tiene éxito. Hemos grabado un vídeo que muestra la reproducción de este test (comentada), y puede verse en `https://www.youtube.com/watch?v=1OBlBKly1pk`.

Las pruebas anteriores ponen de manifiesto que al llevar a cabo la reproducción automática de los test los niveles siguen pudiendo completarse. De esta forma, se demuestra que los diseñadores pueden introducir cambios en el mapa sin romper la estructura del juego, llevando a cabo la reproducción automática de test para comprobar que todo funciona.

5.5.2. Ejemplo de traza

En la figura 5.5 se muestra una pequeña parte del archivo generado cuando el jugador experto juega a Time and Space. Se trata de una traza generada cuando la copia del jugador pulsa un botón. La traza, generada en formato json por su simplicidad y claridad de representación, está dividida en varios json objects, que contienen la siguiente información:

- **info:** En este objeto se muestra toda la información relativa a las entidades asociadas al mensaje. Dentro del mismo aparecen otros tres objetos, que son los siguientes:

  1. **associatedEntity**: Corresponde a la entidad asociada al mensaje. En este caso, la entidad asociada al mensaje *TOUCHED* es la copia del jugador de nombre *PlayerClone1*. Contiene el nombre, la posición de la copia cuando pulsa el botón (representada mediante un Vector3) y el tipo de jugador que ha realizado la acción (podría haber sido el jugador principal en lugar de la copia).

2. **entity**: Indica qué entidad ha sido pulsada, así como su posición y el nombre de la misma.
3. **player**: Contiene la información del jugador principal en este momento del juego. Su posición se usará para intentar aproximar el estado del juego a la hora de reproducir las trazas.

- **timestamp:** Milisegundos en los que ocurre el mensaje, empezando a contar desde el inicio del nivel actual.
- **type:** Tipo del mensaje al cual está asociado la traza. En este caso, *TOUCHED* nos indica que se ha pulsado un botón.

```
{
  "info" : {
    "associatedEntity" : {
      "name" : "PlayerClone1",
      "position" : "Vector3(24.5229, 2.13087e-006, 5.86317)",
      "type" : "PlayerClone"
    },
    "entity" : {
      "name" : "puertaTrigger1",
      "position" : "Vector3(25, 0, 6)",
      "type" : "PuertaTrigger"
    },
    "player" : {
      "name" : "Player",
      "position" : "Vector3(37.0423, -2.24262e-006, -6.53315)",
      "type" : "Player"
    }
  },
  "timestamp" : 73400,
  "type" : "TOUCHED"
}
```

Figura 5.5: Ejemplo de traza generada cuando una copia del jugador pulsa un botón

5.6. Introducción a la automatización de pruebas

La grabación y reproducción de trazas que se ha contado anteriormente tiene como objetivo poder diseñar test similares a los test de unidad, que se puedan automatizar y que sean aplicables a videojuegos. El diseño de esas pruebas, así como los cambios que ha habido que realizar en la máquina de estados de Time and Space para poder lanzar los test directamente se puede

ver en Costero Valero (2015). Aunque todo se explica con detalle en dicha memoria, introducimos aquí la forma de diseñar test utilizando un archivo de texto.

Una vez que es posible reproducir trazas del juego, se puede definir un fichero de entrada con la información de los test que se van a llevar a cabo. En dicho test se pueden incluir distintos parámetros:

- **Objetivos:** el testeador podrá especificar los mensajes que quiere que generen de nuevo y si deben generarse en orden o no. Una vez que los mensajes correspondientes se han vuelto a generar, se puede decir que el objetivo particular se ha cumplido.
- **Tiempo máximo:** a veces no será posible completar alguno de los objetivos, ya que debido a cambios en el mapa será imposible llevar a cabo la acción requerida. Por ese motivo, el testeador puede definir un tiempo máximo para indicar cuándo debe interrumpirse el test si los objetivos no se han satisfecho en ese momento.
- **Fichero de entrada directa del usuario:** se especificará el nombre de los ficheros que contengan las teclas pulsadas por el usuario.
- **Fichero de entrada de acciones del usuario:** nombre de los ficheros que contendrán las acciones a alto nivel llevadas a cabo por el usuario, junto con el momento en el que se produjeron y los atributos de dichas acciones.

Utilizando lo anterior y añadiendo algún detalle más en el fichero de texto, podremos lanzar test y recibir una salida acorde a la reproducción que se ha llevado a cabo. Así, habremos automatizado los test para videojuegos, pudiendo llevarlos a cabo incluso cuando los mapas se han modificado ligeramente y mejorando de esta forma la forma de pasar test que se conocía hasta ahora.

Capítulo 6

Conclusiones y trabajo futuro

Your time is limited, don't waste
it living someone else's life.

Steve Jobs

Resumen: En este último capítulo se resumen los aspectos más importantes de los que se ha hablado previamente en el resto de capítulos del documento. Además, se realiza un estudio de los posibles trabajos futuros que se pueden llevar a cabo relacionados con la automatización de test para videojuegos en función de los resultados obtenidos hasta ahora.

6.1. Conclusiones del proyecto

En los capítulos previos se ha puesto de manifiesto la importancia de mejorar los test de unidad para adaptarlos a la comprobación de la corrección de videojuegos y no sólo a pruebas de software sencillas. Aunque ya existían métodos para automatizar test relacionados con videojuegos, están enfocados principalmente a la comprobación de aspectos del software, sin tener en cuenta que también es necesario comprobar que los mapas y niveles siguen siendo correctos después de realizar cambios. Teniendo en cuenta que los niveles evolucionan al mismo ritmo que la implementación del juego, es necesario encontrar una forma de realizar test automáticos para comprobar que las modificaciones no vuelven el nivel inconsistente.

Por todo lo anterior, en este proyecto hemos implementado la propuesta de llevar a cabo test automáticos para realizar pruebas que van más allá del software, incorporando el diseño como elemento indispensable del desarrollo de todo juego. Para ello nos hemos aprovechado de la estructura que tiene

una arquitectura basada en componentes, que facilita la tarea de introducir nuevos componentes que nos ayuden a cumplir nuestro propósito.

A lo largo de este proyecto se ha explicado cómo grabar trazas de videojuegos implementando un nuevo componente que se encarga de escuchar y grabar todos los mensajes interesantes que le llegan, volcándolos en un fichero de datos que posteriormente se usa para llevar a cabo la reproducción de las trazas.

La grabación de trazas se realiza de dos formas distintas. La primera de ellas consiste en simular la pulsación de teclas por el usuario, y sólo necesita de cuatro funciones auxiliares que se han incorporado al juego para poder reproducir exactamente la misma partida grabada anteriormente. La segunda forma de reproducir trazas utiliza el modelado con redes de Petri para llevar a cabo acciones a alto nivel que están especificadas en uno de los dos ficheros de datos generados previamente. Para ello hace uso de la IA del juego, que guiará al jugador a conseguir sus objetivos utilizando la información disponible en el fichero para buscar los caminos óptimos que le llevan a realizar las acciones necesarias para ganar la partida.

Todo lo anterior se ha probado con una implementación concreta, el juego Time and Space, desarrollado por alumnos del Máster de Videojuegos de la Universidad Complutense de Madrid (Blázquez Checa et al., 2013-2014), cuyos resultados se han expuesto en 5.4. Las pruebas llevadas a cabo han obtenido muy buenos resultados, ya que para los casos de prueba diseñados el jugador ha sido capaz de superar los objetivos establecidos o llegar al final del nivel.

La grabación y reproducción de trazas tiene como objetivo poder llevar a cabo test automáticos que permitan probar juegos sin necesidad de contar con beta testers que jueguen los mismos niveles una y otra vez. La forma en la cual se han llevado a cabo esos test se puede consultar en Costero Valero (2015), que ha llevado a cabo la modificación de la máquina de estados de Time and Space para que sea posible definir ciertos objetivos, lanzar los test de unidad y obtener un resultado de éxito o fallo ante cada prueba realizada.

6.2. Trabajo futuro

A pesar de que el trabajo que hemos llevado a cabo ha puesto de manifiesto importantes resultados relacionados con la automatización de pruebas de videojuegos, sólo ha sido posible probar nuestra solución en un juego real. Sin embargo, aun resta incorporar la implementación de estos nuevos componentes en videojuegos más complejos, con más entidades y acciones más variadas.

En general, los juegos más complejos tendrán implementada una IA más precisa que la usada en Time and Space, por lo que previsiblemente los resultados obtenidos al probarlo en este tipo de juegos serán incluso mejores

que los obtenidos hasta ahora con Time and Space.

Asimismo, también sería interesante implementar algún tipo de interfaz que facilite la escritura de los ficheros de datos a partir de los mensajes que se pueden generar en cada videojuego, ya que hasta el momento lo único que se puede hacer es tener en cuenta diferentes casos en función de los mensajes existentes. Así, si quisiéramos crear algún tipo de mensaje nuevo para incorporarlo a nuestro juego, sería necesario recorrer el código para definir una forma de grabar ese mensaje y cómo llevar a cabo su reproducción posterior, lo cual es algo muy tedioso en caso de que la variedad de las acciones y los mensajes generados sea muy alta.

Apéndice A

Implementación de los módulos extra

A.1. Reproducción directa de trazas: inyección de teclas

```
1  void CInputManager::floodMouseMoved(const GUI::CMouseState
2  &mouseState)
3  {
4      if (!_mouseListeners.empty())
5      {
6          // Actualizamos el estado antes de enviarlo
7          _mouseState.movX = mouseState.movX;
8          _mouseState.movY = mouseState.movY;
9
10         std::list<CMouseListener*>::const_iterator it;
11         it = _mouseListeners.begin();
12         for (; it != _mouseListeners.end(); ++it)
13         {
14             (*it)->mouseMoved(_mouseState);
15         }
16     }
17  } // mouseMoved
18
19 void CInputManager::floodMousePressed(const GUI::CMouseState
20 &mouseState)
21  {
22     if (!_mouseListeners.empty())
23     {
24         // Actualizamos el estado antes de enviarlo
25         _mouseState.button = mouseState.button;
26
27         std::list<CMouseListener*>::const_iterator it;
```

```
28          it = _mouseListeners.begin();
29          for (; it != _mouseListeners.end(); ++it) {
30              (*it)->mousePressed(_mouseState);
31          }
32      }
33  } // mousePressed
34
35  void CInputManager::floodMouseReleased(const GUI::CMouseState
36    &mouseState)
37  {
38      if (!_mouseListeners.empty())
39      {
40          // Actualizamos el estado antes de enviarlo
41          _mouseState.button = mouseState.button;
42
43          std::list<CMouseListener*>::const_iterator it;
44          it = _mouseListeners.begin();
45          for (; it != _mouseListeners.end(); ++it)
46          {
47              (*it)->mouseReleased(_mouseState);
48          }
49      }
50  } // mouseReleased
51
52  void CInputManager::floodKeyDown(GUI::TKey key){
53      if (!_keyListeners.empty())
54          {
55        std::list<CKeyboardListener*>::const_iterator it;
56        it = _keyListeners.begin();
57
58        for (; it != _keyListeners.end(); ++it){
59          if (*it){
60          ((*it))->keyPressed(key);
61          }
62        }
63          }
64  }
65
66  void CInputManager::floodKeyUp(GUI::TKey key){
67      if (!_keyListeners.empty())
68      {
69          std::list<CKeyboardListener*>::const_iterator it;
70          it = _keyListeners.begin();
71          for (; it != _keyListeners.end(); ++it)
72          {
73              if ((*it)){
74                  (*it)->keyReleased(key);
75              }
76          }
```

```
77          }
78      }
```

A.2. Ejemplo de Logger

```
1  #include "Logger.h"
2  #include "../Config.h"
3
4  #include <fstream>
5
6  namespace Trace {
7      CLogger::CLogger(std::string path){
8          _file.open(path, std::ios::out | std::ios::trunc);
9          _lastMessage = "";
10     }
11
12     CLogger::~CLogger(){
13         Json::StyledWriter styledWriter;
14         _file << styledWriter.write(_events);
15         _file.close();
16     }
17
18     void CLogger::saveMessage(Logic::CMessage* msg){
19         std::string txt = " " + msg->getType();
20         saveMessage(txt);
21     }
22
23     void CLogger::saveMessage(std::string &msg){
24         if (msg != _lastMessage) {
25             _events.append(msg);
26             _lastMessage = msg;
27         }
28     }
29
30       // Escribe un mensaje que nos llega en formato json
31     void CLogger::saveMessage(Json::Value msg){
32         if (msg != _lastMessage) {
33             _events.append(msg);
34             Json::StyledWriter styledWriter;
35             std::stringstream ss;
36             ss << styledWriter.write(msg);
37             _lastMessage = ss.str();
38         }
39     }
40
41 } // namespace Trace
```

A.3. Ejemplo de filtro: *SpecialActionFilter*

```
1  #include "Logic\Entity\Message.h"
2  #include "Logic\Entity\Entity.h"
3  #include "ILogger.h"
4  #include "SpecialActionFilter.h"
5  #include <sstream>
6  #include <string>
7  #include <iostream>
8
9
10 namespace Trace{
11   bool CSpecialActionFilter::accept(Logic::CMessage*
12           msg){
13       if (msg->getType()==Logic::TMessageType::SPECIAL_ACTION)
14          return true;
15       return false;
16    }
17
18           // Procesa el mensaje, es decir, en función de la
19           // información que contenga crea un json que contiene
20           // distintos valores
21    void CSpecialActionFilter::processMessage(Logic::CEntity*
22      entity, Logic::CMessage* msg, Json::Value &json, int time){
23       std::stringstream ss;
24
25       Json::Value jsonArray;
26       Json::Value info;
27       json = NULL;
28
29       if (msg->getType()==Logic::TMessageType::SPECIAL_ACTION)
30           {
31          jsonArray["type"] = "SPECIAL_ACTION";
32
33          Logic::CSpecialActionMessage* message =
34                  static_cast<Logic::CSpecialActionMessage*>(msg);
35          std::string subtype;
36          switch (message->_specialAction) {
37          case Logic::TSpecialActions::ACTION:
38             subtype = "ACTION";
39             break;
40          case Logic::TSpecialActions::ACTION_CONFIRMED:
41             subtype = "ACTION_CONFIRMED";
42             break;
43          case Logic::TSpecialActions::GET_OBJECT:
44             subtype = "GET_OBJECT";
45             break;
46          case Logic::TSpecialActions::NONE:
47             subtype = "NONE";
48             break;
49          }
```

```
50
51              jsonArray["subtype"] = subtype;
52
53              info["entity"] = entity->getName();
54              if (message->_additionalInfo != "")
55                  info["additionalInfo"] = message->_additionalInfo;
56
57              Json::Value sender;
58              std::stringstream saux;
59              saux << message->_entity->getPosition();
60              sender["type"] = message->_entity->getType();
61              sender["name"] = message->_entity->getName();
62              sender["position"] = saux.str();
63
64              info["senderEntity"] = sender;
65          }
66
67          if (jsonArray != NULL) {
68              jsonArray["info"] = info;
69              jsonArray["timestamp"] = time;
70
71              json = jsonArray;
72          }
73      }
74
75        // Recibe un mensaje, lo procesa y llama al logger asociado
76        // para que lo escriba en un fichero
77      void CSpecialActionFilter::manageMessage(Logic::CEntity* entity,
78              Logic::CMessage* msg, unsigned long msecs){
79          std::stringstream ss;
80          Json::StyledWriter styledWriter;
81
82          Json::Value jsonArray;
83          processMessage(entity, msg, jsonArray, (int)msecs);
84          if (jsonArray.size() > 0 && jsonArray != NULL) {
85              _logger->saveMessage(jsonArray);
86          }
87      }
88  }
```

Bibliografía

Y así, del mucho leer y del poco dormir,
se le secó el celebro de manera que vino
a perder el juicio.

Miguel de Cervantes Saavedra

Bilas, S. A Data-Driven Game Object System. En *Game Developers Conference*. 2002.

Blázquez Checa, Á., Pérez Alonso, A., Cordón Ureña, A., Bergareche Gros, I., Glaria Abril, J., Pazos Estévez, F. y Rodríguez Zapater, L. I. Time and Space, The Game, `https://timeandspacethegame.wordpress.com/`. 2013-2014.

Costero Valero, L. *Ejecución y adaptación de trazas de juegos para la automatización de pruebas.* . Proyecto Fin de Carrera, Universidad Complutense de Madrid, 2015.

Gómez-Martín, M. A. *Arquitectura y metodología para el desarrollo de sistemas educativos basados en videojuegos*. Phd, Universidad Complutense de Madrid, 2008.

Gómez-Martín, P. P., Llansó, D., Gómez-Martín, M. A., Ontañón, S. y Ram, A. MMPM: a generic platform for case-based planning research. 2009.

Gregory, J. *Game engine architecture*. Taylor and Francis Ltd., 2009.

IEEE Std 829-1998. IEEE Standard for Software Test Documentation. 1998.

Mellon, L. Automatic Testing for Online Games. En *Game Developers Conference*. 2006.

Meszaros, G. *XUnit test patterns: refactoring test code*. Addison-Wesley, 2007.

Myers, G. J. *The Art of Software Testing*. John Wiley and Sons, 1979.